# PLT **mzc**: MzScheme Compiler Manual

Matthew Flatt
mflatt@cs.utah.edu

Sebastian Good
four@rice.edu

PLT
scheme@cs.rice.edu

Version 103
August 2000

Rice University
University of Utah

# Contents

# 1. About **mzc**

## 1.1 **mzc** Is...

The **mzc** compiler takes MzScheme (or MrEd) source code and produces either platform-independent byte code compiled files (**.zo** files) or platform-specific native code libraries (**.so** or **.dll** files) to be loaded into MzScheme (or MrEd).

**mzc** works on either individual files or on collections. (A **collection** is a group of files that conform to MzScheme's library collection system; see §15 in *PLT MzScheme: Language Manual*).

As a convenience for programmers writing low-level MzScheme extensions, **mzc** can compile and link plain C files that use MzScheme's **escheme.h** header. This facility is described in *Inside PLT MzScheme*.

### 1.1.1 Byte Code Compilation

A byte code file typically uses the file extension **.zo**. The file starts with a regular Scheme expression to test MzScheme's version, followed by `#‘` and the bytecode data.

Byte code files are loaded into MzScheme in the same way as regular Scheme source files (i.e., with `load`). The `#‘` marker causes MzScheme's reader to load byte codes instead of normal Scheme expressions.

Byte code programs produced by **mzc** run exactly the same as source code compiled by MzScheme directly (assuming the same set of macros, signatures, and syntax are avilable at compile-time and load-time). In other words, byte code compilation does not optimize the code any more than MzScheme's normal evaluator. However, a byte code file can be loaded into MzScheme much faster than a source code file.

### 1.1.2 Native Code Compilation

A native code file is a platform-specific shared library. Under Windows, native code files typically use the extension **.dll**. Under Unix and MacOS, native code files typically use the extension **.so**.

Native code files are loaded into MzScheme with the `load-extension` procedure (see §14.7 in *PLT MzScheme: Language Manual*).

The native code compiler attempts to optimize a source program so that it runs faster than the source code or byte code version of the program. See §1.5 for information on obtaining the best possible performance from **mzc**-compiled programs.

Native code compilation produces C source code in an intermediate stage; your system must provide an external C compiler to produce native code. The **mzc** compiler cannot produce native code directly from Scheme code.

- Under Unix, **gcc** is used as the C compiler if it can be found in any of the directories listed in the PATH environment variable. If **gcc** is not found, **cc** is used if it can be found.

- Under Windows, **cl.exe**, Microsoft Visual C, is used as the C compiler if it can be found in any of the directories listed in the PATH environment variable. If **cl.exe** is not found, **gcc.exe** is used if it can be found.

- Under MacOS, Metrowerks CodeWarrior is used as the C compiler if it can be found.

Except for MacOS, the C compiler and compiler flags used by **mzc** can be adjusted via command line flags.

## 1.2 mzc Is Not...

**mzc** does not generally produce stand-alone executables from Scheme source code. The compiler's output is loaded into MzScheme (or MrEd or DrScheme). However, see also §4 for information about embedding code into a copy of the MzScheme (or MrEd) executable.

**mzc** does not translate Scheme code into similar C code. Native code compilation produces C code that relies on MzScheme to provide run-time support, which includes memory management, closure creation, procedure application, and primitive operations.

## 1.3 Running mzc

Under Unix and Windows, run **mzc** from a shell, passing in flags and arguments on the command line.

Under MacOS, double-click on the **mzc** launcher application with the Command key pressed, then provide arguments in the command line dialog that appears. (Close the **MzScheme** application first if it is already running, since **mzc** is itself a MzScheme-based application.) If the Command key is not pressed while **mzc** is started, the command-line dialog will not appear. If a file is dragged onto the **mzc** icon, then the command-line will contain the file's path; this is useful for compiling a Scheme file directly to an extension. If a file is dragged onto the **mzc** icon, additional command-line argument can be provided by holding down the Command key, but the arguments will go after the file name, which is almost never useful (since the order of command-line arguments is important).

In this manual, each example command line is shown as follows:

```
mzc --extension --prefix macros.ss file.ss
```

To run this example under Unix or Windows, type the command line into a shell (replacing **mzc** with the path to **mzc** on your system, if necessary). Under MacOS, launch **mzc** with the Command key pressed, and enter everything *after* **mzc** into the dialog that appears.

Simple on-line help is available for **mzc**'s command-line arguments by running **mzc** with the `-h` or `--help` flag.

## 1.4 Macros and Signatures in mzc

When **mzc** compiles a Scheme file, macros and signatures are expanded away. Macro and signature definitions in a source file are evaluated (so the macros and signatures can be used later in the source file), but the definitions are not preserved, so loading the compiled file will not redefine the macros or signatures.

Elaboration-time expressions (i.e., `begin-elaboration-time` and `begin-construction-time` expressions) are also evaluated by compilation; the S-expression result of an elaboration-time expression is compiled in place of the elaboration-time expression.

## 1.5 Native Code Optimization from mzc

Compiling a program to native code with **mzc** can provide significant speedups compared to interpreting byte codes (or running the program directly from source code) for certain kinds of programs. The speedup from native code compilation is typically due to two optimizations:

- **Loop Optimization** — When **mzc** statically detects a tail-recursive loop, it compiles the Scheme loop to a C loop that has no interpreter overhead. For example, given the program

  ```
  (letrec ([odd (lambda (x)
                  (if (zero? x)
                      #f
                      (even (sub1 x))))]
           [even (lambda (x)
                   (if (zero? x)
                       (odd (sub1 x))))])
    (odd 40000))
  ```

  **mzc** can detect the `odd`–`even` loop and produce native code that runs twice as fast as byte code interpretation. In contrast, given a similar program using top-level definitions,

  ```
  (define (odd x) ...)
  (define (even x) ...)
  ```

  the compiler cannot assume an `odd`–`even` loop, because the global variables `odd` and `even` can be redefined at any time. Note that `define`d variables in a `unit` module are lexically scoped like `letrec` variables, and `unit` definitions therefore permit loop optimizations.[1]

- **Primitive Inlining** — When **mzc** encounters the application of certain primitives, it inlines the primitive procedure. However, the compiler must be certain that a variable reference will resolve to a primitive procedure when the code is loaded into MzScheme. In the preceding example, the compiler cannot inline the application of `sub1` because the global variable `sub1` might be redefined. To encourage the inlining of primitives—which produces native code that runs *30 times faster* than byte code interpretation for the preceding example—the programmer has three options:
  - **Use #% keywords** — If `sub1` is replaced in the preceding example by `#%sub1`, `add1` is replaced by `#%add1`, and `zero?` is replaced by `#%zero?`, then the compiler inlines each of the primitives.
  - **Use the `--prim` flag** — The `--prim` flag alters the semantics of the langugage for compilation such that every reference to a global variable that is built into MzScheme is converted to its keyword form. Thus, specifying the `--prim` flag causes **mzc** to automatically convert `sub1` to `#%sub1`, etc.
  - **Use units (MzScheme modules)** — If the original example is encapsulated in a unit, then each primitive name, such as `sub1`, is guranteed to access the primitive procedure (assuming that the name is not lexically bound). The "unitized" version of the preceding program follows:

    ```
    (define oe-unit
      (unit (import) (export) ; import nothing, export nothing
        (letrec ([odd (lambda (x)
                        (if (zero? x)
                            #f
                            (even (sub1 x))))]
    ```

[1] The compiler cannot always prove that `unit` definitions have been evaluated before the corresponding variable is used in an expression. Use the `-v` or `--verbose` flag to check whether **mzc** reports a "last known unit binding" warning when compiling a `unit` expression, which indicates that definitions after a particular line in the source file might be referenced before they are defined.

```
             [even (lambda (x)
                     (if (zero? x)
                         (odd (sub1 x))))])
      (odd 40000))))
(invoke-unit oe-unit) ; run the program
```

Programs that permit these optimizations also to encourage a host of other optimizations, such as procedure inlining (for programmer-defined procedures) and static closure detection. In general, `unit`-based programs provide the most opportunities for optimization.

Native code compilation rarely produces significant speedup for programs that are not loop-intensive, programs that are heavily object-oriented, programs that are allocation-intensive, or programs that exploit built-in procedures (e.g., list operations, regular expression matching, or file manipulations) to perform most of the program's work.

# 2. Compiling Individual Files with **mzc**

To compile an individual file with **mzc**, provide the file name as the command line argument to **mzc**. To compile to byte code, use the `-z` or `--zo` flag; to compile to native code, use the `-e` or `--extension` flag. If no compilation mode flag is specified, `--extension` is assumed.

The input file must have a file extension that designates it as a Scheme file, either **.ss** or **.scm**. The output file will have the same base name and same directory (by default) as the input file, but with an extension appropriate to the type of the output file (either **.zo**, **.dll**, or **.so**).

Example:

```
mzc --extension file.ss
```

Under Windows, the above command reads **file.ss** from the current directory and produces **file.dll** in the current directory.

Multiple Scheme files can be specified for compilation at once. A separate compiled file is produced for each Scheme file. By default, each compiled file is placed in the directory containing the corresponding input file. When multiple files are compiled at once, macros defined in a file are visible in the files that are compiled afterwards.

## 2.1 Prefixing Compilation with Macro and Signature Definitions

A `load` or `require-library` expression in a source file is compiled—but *not evaluated!*—as the source file is compiled. Even if the `load` or `require-library` expression loads macro or signature definitions, these will not be loaded as the file is compiled. To fix this problem for macro- and signature-defining files and libraries, wrap each `load` or `require-library` expression with `(begin-elaboration-time ...)`, which directs the compiler to evaluate the wrapped expression at compilation time.

For example, suppose that **x.ss** contains the following Scheme code:

```
(require-library "macro.ss")
(define f (opt-lambda () 10))
```

If **x.ss** is loaded directly into MzScheme, `f` is defined as expected. But if **x.ss** is compiled to **x.so** using **mzc**, and then **x.so** is loaded into MzScheme, the result is an "expected procedure, given #<macro>" exception. The problem is that **macro.ss**, which defines tha `opt-lambda` macro, is not loaded until run-time, when **x.so** is loaded. As a result, the `(opt-lambda () 10)` expression is compiled as a procedure application. To correct the problem, we wrap the `require-library` expression with `begin-elaboration-time`, which instructs **mzc** to load the library at compile-time:

```
(begin-elaboration-time (require-library "macro.ss"))
(define f (opt-lambda () 10))
```

**mzc**'s `-p` or `--prefix` flag takes a file and loads it at elaboration time before compiling the source files specified on the command line. This is useful for installing a set of macros or signatures that the source files expect to be present already.

## 2.2 Autodetecting Compiled Files for Loading

When MzScheme's `load/use-compiled`, `load-relative`, or `require-library` procedure is used to load a file, MzScheme automatically detects an alternate byte code and/or native code compiled file that resides near the requested file. Byte code files are found in a **compiled** subdirectory in the directory of the requested file. Native code files are found in `(build-path` *dir* `"compiled" "native" (system-library-subpath))` where *dir* is the directory of the requested file. A byte code or native code file is used in place of the rquested file only if its modification date is later than the requested file. If both byte code and native code files are found, the native code file is loaded.

Example:

```
mzc --extension --destination compiled/native/i386-linux file.ss
```

Under Linux, the above command compiles **file.ss** in the current directory and produces **compiled/native/i386-linux/file.so**. Evaluating `(load/use-compiled "file.ss")` in MzScheme will then load **compiled/native/i386-linux/file.so** instead of **file.ss**. If **file.ss** is changed without recreating **file.so**, then `load/use-compiled` loads **file.ss**, because **file.so** is out-of-date.

## 2.3 Compiling Multiple Files to a Single Native Code Library

When the `-o` or `--object` flag is provided to **mzc**, **.kp** and **.o**/**.obj** files are produced instead of a loadable library. The **.o**/**.obj** files contain the native code for a single source file. The **.kp** files contain information used for global optimizations.

Multiple **.kp** and **.o**/**.obj** files are linked into a single library using **mzc** with the `-l` or `--link-extension` flag. All of the **.kp** and **.o**/**.obj** files to be linked together are provided on the command line to **mzc**. The output library is always named **_loader.so** or **_loader.dll**.

Example:

```
mzc --object file1.ss
mzc --object file2.ss
mzc --link-extension file1.kp file1.o file2.kp file2.o
```

Under Unix, the above commands produce a **_loader.so** library that encapsulates both **file1.ss** and **file2.ss**.

Loading **_loader** into MzScheme is not quite the same as loading all of the Source files that are encapsulated by **_loader**. The return value from `(load-extension "_loader.so")` is a procedure that takes a symbol or `#t`. If a symbol is provided and it is the same as the base name of a source file (i.e., the name without a path or file extension) encapsulated by **_loader**, then a thunk is returned. Applying this thunk has the same effect as loading the corresponding source file. If a symbol is not recognized by the **_loader** procedure, `#f` is returned instead of a thunk. If `#t` is provided, a thunk is returned that "loads" all of the files (using the order of the **.o**/**.obj** files provided to **mzc**) and returns the result from loading the last one.

The **_loader** procedure can be called any number of times to obtain thunks, and each thunk can be applied any number of times (where each application has the same effect as loading the source file again). Evaluating `(load-extension "_loader.so")` multiple times returns an equivalent loader procedure each time.

Given the **_loader.so** constructed by the example commands above, the following Scheme expressions have the same effect as loading **file1.ss** and **file2.ss**:

```
(((load-extension "_loader.so") 'file1))
(((load-extension "_loader.so") 'file2))
```

or, equivalently:

```
(((load-extension "_loader.so") #t))
```

The special **_loader** convention is recognized by MzScheme's `load/use-compiled`, `load-relative`, and `require-library` procedures. MzScheme automatically detects **_loader.so** or **_loader.dll** in the same directory as individual native code files (see §2.2). If both an individual native code file and a **_loader** are available, the **_loader** file is used.

# 3. Compiling Collections with **mzc**

A **collection** is a group of files that conform to MzScheme's library collection system; see §15 in *PLT MzScheme: Language Manual* for details.

The `--collection-zo` and `--collection-extension` flags direct **mzc** to compile a whole collection. The `--collection-zo` flag produces individual **.zo** files for each library in the collection. The `--collection-extension` flag produces a single **_loader** library for the collection.

The (sub-)collection to compile is specified on the command line for **mzc**. The specified collection must contain an **info.ss** library that provides information about how to compile the collection. The result of loading the **info.ss** library must be a procedure that takes two arguments: a symbol and a failure thunk. The symbol specifies a requested **field**, i.e., the kind of information is requested. If the requested information is available, then the **info.ss** procedure returns that information; otherwise, it must call the failure thunk.

For example, the following procedure is in the **info.ss** library of the **help** collection:

```
(lambda (request failure-thunk)
  (case request
    [(name) "Help"]
    [(compile-prefix) `(begin
                         (require-library "sig.ss" "mred")
                         (require-library "sig.ss" "help"))]
    [(compile-omit-files) (list "sig.ss" "manuals.ss")]
    [(compile-elaboration-zos) (list "sig.ss")]
    [(mred-launcher-libraries) (list "help.ss")]
    [(mred-launcher-names) (list "Help Desk")]
    [else (failure-thunk)]))
```

This example **info.ss** procedure provides information for six fields: `'name`, `'compile-prefix`, etc.

The **info.ss** system is standardized by convention. A collection's **info.ss** file can be used by several clients (including **mzc**, **Setup PLT**, and **Help Desk**), each requesting a different set of fields.

To compile a collection, **mzc** extracts **info.ss** information for the following fields:

- `'name` — the name of the collection as a string.
- `'compile-prefix` — an S-expression to use as the elaboration-time prefix expression for compiling all files in the collection. This information is optional (i.e., the failure thunk can be called when this symbol is provided to the **info.ss** procedure), but it is recommended beacuse it indicates to external tools that the collection can be compiled. (Use `'(void)` if no prefix is needed.)
- `'compile-omit-files` — a list of library filenames (without paths); all Scheme files in the collection are compiled except for the files in this list. If a library contains elaboration time expressions (e.g., macros or signatures) that are not local to the file, then the library file should be included in this list. This information is optional.

- `'compile-zo-omit-files` — a list of library filenames that should not be compiled to byte code (but possibly to native code). This information is optional.
- `'compile-extension-omit-files` — a list of library filenames that should not be compiled to native code (but possibly to byte code). This information is optional.
- `'compile-subcollections` — a list of sub-collection sub-paths, where each sub-path is a list of strings; each full sub-collection path is formed by appending the sub-path to the path of the collection being compiled. Each sub-collection is compiled in the same way as the current collection, using the **info.ss** library of the sub-collection. This information is optional.

When compiling a collection to byte code files, **mzc** automatically creates a **compiled** directory in the collection directory and puts **.zo** files there.

When compiling a collection to native code, **mzc** automatically created a **compiled** directory in the collection directory, a **native** directory in that **compiled** directory, and a platform-specific directory in **native** using the directory name returned by `system-library-subpath`. Intermediate **.c** and **.kp** files are kept in **native**. The platform-specific directory gets intermediate **.o**/**.obj** files and the final **_loader.so** or **_loader.dll**.

To compile a collection, **mzc** compiles only the library files that have changed since the last compilation. This form of dependency-checking is usually too weak. For example, when a signature file changes, **mzc** does not automatically recompile all files that rely on the signatures. In this case, delete the **compiled** directory when a macro or signature file changes to ensure that the collection is compiled correctly.

## 3.1 Macro and Elaboration-Time Libraries

Libraries that define macros or signatures cannot be compiled to native code. These libraries can be "compiled" to byte code by listing them in a special **info.ss** field:

- `'compile-elaboration-zos` — a list of library filenames (without paths); the library files in this list are compiled with macros, signatures, and elaboration expressions both evaluated and preserved in the **.zo** file. (There is one exeception: expressions using `#%begin-elaboration-time` instead of `begin-elaboration-time` are not preserved.)

  The files in the list are compiled from left to right, all in the same namespace. If a file does not need to be compiled, it is nevertheless loaded before subsequent files in the list are compiled, so macros and signatures defined by the file are available for later files.
- `'compile-elaboration-zos-prefix` — an S-expression to use as an elaboration-time prefix expression for compiling the files returned for `'compile-elaboration-zos`.

# 4. Building a Stand-alone Executable

Since the output of **mzc** relies on MzScheme to provide all run-time support, there is no way to use **mzc** to obtain *small* stand-alone executables. However, it is possible to produce a *large* stand-alone executable that contains an embedded copy of the MzScheme (or MrEd) run-time engine.

## 4.1 Stand-Alone Executables from Scheme Code

The command-line flags `--exe` directs **mzc** to embed Scheme source or byte code into a copy of the MzScheme executable. The `--gui-exe` flag is similar, but copies the MrEd executable.

The embedding operation merely merges the given Scheme code into the executable. If the embedded code refers to MzLib libraries or other collection files, the libraries must still be present when the stand-alone executable is run (and the executable must be able to find the library collections in the usual way).

## 4.2 Stand-Alone Executables from Native Code

Creating a stand-alone executable that embeds native code from **mzc** requires downloading the MzScheme source code and using a C compiler and linker directly.

To build an executable with an embedded MzScheme engine:

- Download the source code from `http://www.cs.rice.edu/CS/PLT/packages/mzscheme` and compile MzScheme.
- Recompile MzScheme's **main.c** with the preprocessor symbol `STANDALONE_WITH_EMBEDDED_EXTENSION` defined. Under Unix, the **Makefile** distributed with MzScheme provides a target **ee-main** that performs this step.

  The preprocessor symbol causes MzScheme's startup code to skip command line parsing, the user's initialization file, and the `read-eval-print` loop. Instead, the C function `scheme_initialize` is called, which is the entry point into **mzc**-compiled Scheme code. After compiling **main.c** with `STANDALONE_WITH_EMBEDDED_EXTENSION` defined, MzScheme will not link by itself; it must be linked with objects produced by **mzc**.
- Compile each Scheme source file in the program with **mzc**'s `-o` or `--object` flag and the `--embedded` flag, producing a set of **.kp** files and object (**.o** or **.obj**) files.
- After each Scheme file is compiled, run **mzc** with the `-g` or `--link-glue` and the `--embedded` flag, providing all of the **.kp** files and object files on the command line. (Put the object files in the order that they should be "loaded.") The `-g` or `--link-glue` step produces a new object file, **_loader.o** or **_loader.obj**.

  Each of the Scheme source files in the program must have a different base name (i.e., the file name without its directory path or extension), otherwise **_loader** cannot distinguish them. The files need not reside in the same directory.

- Link all of the **mzc**-created object files with the MzScheme implementation (having compiled **main.c** with `STANDALONE_WITH_EMBEDDED_EXTENSION` defined) to produce a stand-alone executable.

  Under Unix, the **Makefile** distributed with MzScheme provides a target **ee-app** that performs the final linking step. To use the target, call **mzmake** with a definition for the makefile macro **EEAPP** to the output file name, and a definition for the makefile macro **EEOBJECTS** to to the list of **mzc**-created object files. (The example below demonstrates how to define makefile variables on the command line.)

For example, under Unix, to create a standalone executable **MyApp** that is equivalent to

```
mzscheme -mv -f file1.ss -f file2.ss
```

unpack the MzScheme source code and perform the following steps:

```
cd plt/src/mzscheme
./mzmake
./mzmake ee-main
mzc --object --embedded file1.ss
mzc --object --embedded file2.ss
mzc --link-glue --embedded file1.kp file1.o file2.kp file2.o
./mzmake EEAPP=MyApp EEOBJECTS="file1.o file2.o _loader.o" ee-app
```

To produce an executable that embeds the MrEd engine, the procedure is essentially the same. MrEd is compiled somewhat differently from MzScheme (e.g., there's no **mzmake**), and MrEd's main file is **mred.cxx** instead of **main.c**. See the compilation notes in the MrEd source code distribution for more information.

# Index